

# Oracular Form and Computing Strong Game-Theoretic Jotto Strategies

Sam Ganzfried

Carnegie Mellon University  
Computer Science Department  
sganzfri@cs.cmu.edu

**Abstract.** We develop a new approach that computes approximate equilibrium strategies in Jotto, a popular word game. Jotto is an extremely large two-player game of imperfect information; its game tree has many orders of magnitude more states than games previously studied, including no-limit Texas Hold'em poker. To address the fact that the game is so large, we propose a novel strategy representation called oracular form, in which we do not explicitly represent a strategy, but rather appeal to an oracle that quickly outputs a sample move from the strategy's distribution. Our overall approach is based on an extension of the fictitious play algorithm to this oracular setting. We make several interesting observations from the computed strategies and demonstrate the superiority of our algorithm over a benchmark algorithm.

## 1 Introduction

Developing strong strategies for agents in large games is an important problem in artificial intelligence. In particular, much work has been devoted in recent years to developing algorithms for computing game-theoretic solution concepts, specifically the Nash equilibrium. In two-player zero-sum games, Nash equilibrium strategies have a strong theoretical justification as they also correspond to minimax strategies; by following an equilibrium strategy, a player can guarantee at least the value of the game in expectation, regardless of the strategy followed by his opponent. Currently, the best algorithms for computing a Nash equilibrium in two-player zero-sum extensive-form games (with perfect recall) are able to solve games with  $10^{12}$  states in their game tree [10].

Unfortunately, many interesting games actually have significantly more than  $10^{12}$  states. Texas Hold'em poker is a prime example of such a game that has received significant attention in the AI literature in recent years; the game tree of two-player limit Texas Hold'em has about  $10^{18}$  states, while that of two-player no-limit Texas Hold'em has about  $10^{71}$  states. The standard approach of dealing with this is to apply an *abstraction* algorithm, which constructs a smaller game that is similar to the original game; then the smaller game is solved, and its solution is mapped to a strategy profile in the original game [1]. Many abstraction algorithms work by coarsening the moves of chance, collapsing

several information sets of the original game into single information sets of the abstracted game (called *buckets*).

In this paper we study a game with many orders of magnitude more states than even two-player no-limit Texas Hold'em. Jotto, a popular word game, contains approximately  $10^{868}$  states in its game tree (and far more if we are not clever about how the game is represented). Unfortunately, Jotto does not seem particularly amenable to abstraction in the same way that poker is; we discuss reasons for this in Section 4. Furthermore, even if we could apply an abstraction algorithm to Jotto, we would need to group  $10^{856}$  game states into a single bucket on average, which would almost certainly lose a significant amount of information from the original game. Thus, the abstraction paradigm that has been successful on poker does not seem promising to games like Jotto; an entirely new approach is needed.

We provide such an approach. To deal with the fact that we cannot even represent a strategy for one of the players, we provide a novel strategy representation which we call *oracular form*. Rather than viewing a strategy as an explicit object that must be represented and stored, we instead represent it implicitly through an oracle; we can think of the oracle as an efficient algorithm. Each time we want to make a play from the strategy, we query the oracle, which quickly outputs a sample play from the strategy's distribution. Thus, instead of representing the entire strategy in advance, we obtain it on an as-needed basis via real-time computation.

Our main algorithm for computing an approximate equilibrium in Jotto is an extension of the fictitious play algorithm [3] to our oracular setting. The algorithm outputs a full strategy for one player, and for the other player outputs data such that if another algorithm is run on it, a sample of the strategy's play is obtained. Thus, we can play this strategy, even though it is never explicitly represented. We use our algorithm to compute approximate equilibrium strategies on 2, 3, 4, and 5-letter variants of Jotto. We also make some interesting observations concerning the equilibria of these four games, as well as demonstrate the superiority of our algorithm over a benchmark algorithm.

## 2 Game theory background

In this section, we review relevant definitions and prior results from game theory and game solving.

### 2.1 Strategic-form games

The most basic game representation, and the standard representation for simultaneous-move games, is the *strategic form*. A *strategic-form game* (aka matrix game) consists of a finite set of players  $N$ , a space of *pure strategies*  $S_i$  for each player, and a utility function  $u_i : \times_i S_i \rightarrow \mathbb{R}$  for each player. Here  $\times_i S_i$  denotes the space of *strategy profiles* — vectors of pure strategies, one for each player.

The set of *mixed strategies* of player  $i$  is the space of probability distributions over his pure strategy space  $S_i$ . We will denote this space by  $\Sigma_i$ . If the sum of the payoffs of all players equals zero at every strategy profile, then the game is called *zero sum*. In this paper, we will be primarily concerned with two-player zero-sum games. If the players are following strategy profile  $\sigma$ , we let  $\sigma_{-i}$  denote the strategy taken by the opponent.

## 2.2 Extensive-form games

An *extensive-form* game is a general model of multiagent decision-making with potentially sequential and simultaneous actions and imperfect information. As with perfect-information games, extensive-form games consist primarily of a game tree; each non-terminal node has an associated player (possibly *chance*) that makes the decision at that node, and each terminal node has associated utilities for the players. Additionally, game states are partitioned into *information sets*, where the player whose turn it is to move cannot distinguish among the states in the same information set. Therefore, in any given information set, the player whose turn it is to move must choose actions with the same distribution at each state contained in the information set. If no player forgets information that he previously knew, we say that the game has *perfect recall*.

## 2.3 Mixed vs. behavioral strategies

There are multiple ways of representing strategies in extensive-form games. Define a *pure strategy* to be a vector that specifies one action at each information set. Clearly there will be an exponentially number of pure strategies in the size of the game tree; if the tree has  $I_i$  information sets for player  $i$  and  $A_i$  possible actions at each information set, then player  $i$  has  $A_i^{I_i}$  possible pure strategies. Define a *mixed strategy* to be a probability distribution over the space of pure strategies. We can represent a mixed strategy as a vector with  $A_i^{I_i}$  components.

Alternatively, we could play a strategy that randomizes independently at each information set; we refer to such a strategy as a *behavioral strategy*. Since a behavioral strategy must specify a probability for playing each of  $A_i$  actions at each information set, we can represent it as a vector with only  $A_i \cdot I_i$  components. Thus, behavioral strategies can be represented exponentially more compactly than mixed strategies.

Fortunately, it turns out that this gain in representation size does not come at the loss of expressiveness; that is, any mixed strategy can also be represented as an equivalent behavioral strategy (and vice versa) [5]. Thus, all current computational approaches to extensive-form games operate on behavioral strategies and avoid the unnecessary exponential blowup associated with using mixed strategies.

## 2.4 Nash equilibria

Player  $i$ 's *best response* to  $\sigma_{-i}$  is any strategy in  $\arg\max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i})$ . A *Nash equilibrium* is a strategy profile  $\sigma$  such that  $\sigma_i$  is a best response to  $\sigma_{-i}$ .

for all  $i$ . An  $\epsilon$ -*equilibrium* is a strategy profile in which each player achieves a payoff of within  $\epsilon$  of his best response.

In two player zero-sum games, we have the following result which is known as the *minimax theorem*:

$$v^* = \max_{\sigma_1 \in \Sigma_1} \min_{\sigma_2 \in \Sigma_2} u_1(\sigma_1, \sigma_2) = \min_{\sigma_2 \in \Sigma_2} \max_{\sigma_1 \in \Sigma_1} u_1(\sigma_1, \sigma_2).$$

We refer to  $v^*$  as the *value* of the game to player 1. Sometimes we will write  $v_i$  as the value of the game to player  $i$ . Any equilibrium strategy for a player guarantees an expected payoff of at least the value of the game to that player.

All finite games have at least one Nash equilibrium. Currently, the best algorithms for computing a Nash equilibrium in two-player zero-sum extensive-form games with perfect recall are able to solve games with  $10^{12}$  states in their game tree [10].

### 3 Smoothed fictitious play

In this section we will review the fictitious play (FP) algorithm [3]. Despite being quite simple, FP is the state-of-the-art algorithm for solving many classes of games [4,7,9]. The rest of this section will assume the game has two players (though FP has been successfully applied to games with many players).

The basic FP algorithm works as follows; at each iteration, each player plays a best response to the average strategy of his opponent so far. Formally, in smoothed fictitious play (SFP) each player  $i$  applies the following update rule at each time step  $t$ :

$$s_{i,t} = \left(1 - \frac{1}{t+1}\right) s_{i,t-1} + \frac{1}{t+1} s_{i,t}^{BR},$$

where  $s_{i,t}^{BR}$  is a best response of player  $i$  to the strategy  $s_{-i,t-1}$  of his opponent at time  $t-1$ . We allow strategies to be initialized arbitrarily at  $t=0$ .

FP is guaranteed to converge to a Nash equilibrium in two-player zero-sum games; however, very little is known about how many iterations are needed to obtain convergence. Recent work shows that FP may require exponentially many iterations in the worst case [2]; however, it may perform far better in practice on specific games. In addition, the performance of FP is not monotonic; for example, it is possible that the strategy profile after 200 iterations is actually significantly closer to equilibrium than the profile after 300 iterations. So simply running FP for some number of iterations and using the final strategy profile is not necessarily the best approach.

We instead use the following improved algorithm. For each iteration we compute the amount each player could gain by deviating to a best response; denote it by  $\epsilon_{i,t}$ . Let  $\epsilon_t = \max_i \epsilon_{i,t}$ , and let  $\epsilon_{t^*} = \min_{0 \leq t \leq T} \epsilon_t$ . After running FP for  $T$  iterations, rather than output  $s_{i,T}$ , we will instead output  $s_{i,t^*}$  — the  $\epsilon$ -equilibrium for smallest  $\epsilon$  out of all the iterations of FP so far.

## 4 Jotto

Jotto is a popular two-player word game. While there are many different variations of the game, we will describe the rules of one common variant [6]. Each player picks a secret five-letter word, and the object of the game is to correctly guess the other player's word first. Players take turns guessing the other player's secret word and replying with the number common letters between the guessed word and the secret word (the positions do not matter). For example, if the secret word is GIANT and a player guesses PECAN, the other player will give a reply of 2 (for the A and the N, even though they are in the wrong positions). Players often cross out letters that are eliminated and record other logical deductions on a scrap piece of paper.

Instead of having both players simultaneously guessing the other player's word, we could instead just have one player pick a secret word while the other player guesses it. Let us refer to these players as the *hider* and the *guesser* respectively. If the guesser correctly guesses the word on his  $k$ 'th try, then the hider gets payoff  $k$  while the guesser gets payoff  $-k$ . This is the variation that we consider in the remainder of the paper.

There are a few limits on the words that the players can select. All words must be chosen from a pre-arranged dictionary. No proper nouns are allowed, and the words must consist of all different letters (some variations do not impose this restriction). Furthermore, we do not allow players to select a word of which other permutations (aka anagrams) are also valid words (e.g., STARE and RATES)<sup>1</sup>.

The official dictionary we will be using has 2833 valid 5-letter words. A naïve attempt at determining the size of the game tree is the following. At the  $k$ -th guess, the guesser must choose from  $2834 - k$  words. Then he is given one of 6 possible answers for the number of correct letters (since an answer of 5 means the game is over, only the answers 0-4 will lead to further branching in the tree). The total number of states in the tree will be approximately

$$2833 \cdot 2832^5 \cdot 2831^5 \cdot \dots \cdot 1 = 2833 \cdot (2832!)^5. \quad (1)$$

It turns out that we can represent the game much more concisely if we take advantage of the fact that many paths of play lead to the guesser knowing the exact same information. For example, if the player guesses GIANT and gets reply of 2 followed by guessing PECAN with a reply of 3, he knows the exact same information as if he had guessed PECAN with a reply of 3 followed by GIANT with a reply of 2; both sequences should lead to the same game state. More generally, two sequences of guesses and answers lead to identical knowledge bases if and only if the set of words consistent with both sequences is identical. Thus, there is a one-to-one correspondence between a knowledge base of the guesser and a subset of the set of words in the dictionary corresponding to the set of words that are consistent with the guesses so far. Since there are  $2^{2883}$  total subsets of words in the dictionary, the total number of game states where

---

<sup>1</sup> If this restriction were not imposed, then players might need to guess all possible permutations of a word even once all the letters are known.

the guesser would need to make a decision is  $2^{2883} \approx 10^{868}$ . Since the best equilibrium-finding algorithms can only solve games with up to  $10^{12}$  nodes, we have no hope of solving Jotto by simply applying an existing algorithm.

Furthermore, Jotto does not seem amenable to the same abstraction paradigm that has been successful on poker. In poker, abstraction works by grouping several states into the same *bucket* and forcing all states in the same bucket to follow the same strategy. In our compact representation of Jotto, the states correspond to subsets of the dictionary; abstraction would mean that several subsets are grouped together into single buckets. However, the action taken at each bucket will be a single word (i.e., the next guess). If many subsets are grouped together into the same bucket, then there will clearly be some words that have already been guessed in some states in the bucket, while not in other states. This will lead to certain actions being ill-defined, as well as possible infinite loops in the structure of the abstracted game tree. This will be exacerbated by the fact that many states will need to be grouped into the same bucket; on average, each bucket will contain  $10^{856}$  game states.

In short, there are significant challenges that must be overcome to apply any sort of abstraction to Jotto; this may not even be feasible to do at all. Instead, we propose an entirely new approach which we describe next.

## 5 Oracular strategies

Consider the following scenario. Suppose one is playing an extensive-form game  $G$  with  $2^{100}$  information sets and two actions per information set, and that he wishes to play an extremely simple strategy: always choose the first action at each information set (suppose actions are labeled as Action 1 and Action 2). To represent this pure strategy, technically we must list out a vector of size  $2^{100}$  (with each entry being a 1 for this particular strategy). On the other hand, it is trivial to write an algorithm that takes as input the index of an information set and outputs the action taken by this strategy (i.e., output 1 on all inputs). Even though there are a large number of information sets, we only require 100 bits to represent the index of each one; thus, it is possible to play this simple strategy without ever explicitly representing it.

More generally, let  $O_i$  be an efficient deterministic algorithm that takes as input the index of an information set  $I$  and outputs an action from  $A_I$ , the set of actions available at  $I$ . We refer to  $O_i$  as a *pure oracular strategy* for player  $i$ . It is easy to see that every pure oracular strategy is strategically equivalent to a pure strategy  $s_i$  of the game; at information set  $I$ ,  $s_i$  simply plays whatever action  $O_i$  outputs on input  $I$ .

We define oracular versions of randomized strategies analogously to the extensive-form case. Let  $\{O_i\}$  be a collection of pure oracular strategies; then any probability distribution over elements of  $\{O_i\}$  is a *mixed oracular strategy*. If we let  $O_i$  be a randomized algorithm that outputs a probability distribution over actions at each information set, then call  $O_i$  a *behavioral oracular strategy*.

As was the case with pure strategies, each oracular strategy corresponds to a single extensive-form strategy of the same type.

## 6 Computing best responses in Jotto

In order to apply smoothed fictitious play to Jotto, we must figure out how to compute a best response for each player. This turns out to be tricky for a few reasons. First, the guesser’s strategy space is so large that we cannot compute a full best response; we must settle for computing an approximate best response, which we call the guesser’s *greedy best response*. In addition, we represent the guesser’s strategy in oracular form; so the hider cannot operate on it explicitly, and can only query it at certain game states. It turns out that we can actually compute an exact best response for the hider despite this limitation.

### 6.1 Computing the guesser’s greedy best response

Suppose we are given a strategy for the hider, and wish to compute a counter-strategy for the guesser. Let  $h$  denote the strategy of the hider, where  $h_i$  denotes the probability that the hider chooses  $w_i$  — the  $i$ ’th word in the dictionary. Let  $D$  denote the number of words in the dictionary, and let  $S$  be a bit-vector of size  $D$ , where  $S_i = 1$  means that  $w_i$  is still consistent with the guesses so far. So  $S$  encodes the current knowledge base of the guesser and represents the state of the game.

A reasonable heuristic to use for the guesser would be the following. For each word  $w_i$  in the dictionary, compute the number of words that we will eliminate in expectation (over  $h$ ) if we guess  $w_i$ . Then guess the word that expects to eliminate the most words. We refer to this algorithm as GuesserGBR (for “Greedy Best Response”); pseudocode is given in Algorithm 1.

GuesserGBR relies on a number of subroutines. `ExpNumElims`, given in Algorithm 2, gives the expected number of words eliminated if  $w_i$  is guessed. `AnswerProbs`, given in Algorithm 4, gives a vector of the expected probability of receiving each answer from the hider when  $w$  is guessed. `NumElims`, given in Algorithm 3, gives the number of words that can be eliminated when  $w_i$  is guessed and an answer of  $j$  is given. Finally, `NumCommLetts`, given in Algorithm 5, gives the number of common letters between two words. In the pseudocode,  $L$  denotes the number of letters allowed per word. For efficiency, we precompute a table of size  $D^2$  storing all the numbers of common letters between pairs of words. The overall running time of GuesserGBR is  $O(D^2L)$ .

It is worth noting that the greedy best response is not an actual best response; it is akin to searching one level down the game tree and then using the evaluation function “expected number of words eliminated” to determine the next move. This is essentially 1-ply minimax search. While we would like to compute an exact best response by searching the entire game tree, this is not feasible since the tree has  $10^{868}$  nodes. As with computer chess programs, we will need to settle for searching down the tree as far as we can, then applying a reasonable evaluation function.

---

**Algorithm 1** GuesserGBR( $h, S$ )

---

```
for  $i = 1$  to  $D$  do
   $n_i \leftarrow \text{ExpNumElims}(w_i, h, S)$ 
end for
return  $w_i$  with maximal value of  $n_i$ 
```

---

---

**Algorithm 2** ExpNumElims( $w_i, h, S$ )

---

```
 $A \leftarrow \text{AnswerProbs}(w_i, h, S)$ 
 $n \leftarrow 0$ 
for  $j = 0$  to  $L$  do
   $n \leftarrow n + A_j \cdot \text{NumElims}(w_i, S, j)$ 
end for
return  $n$ 
```

---

---

**Algorithm 3** NumElims( $w_i, S, j$ )

---

```
counter  $\leftarrow 0$ 
for  $k = 1$  to  $D$  do
  if  $S_k = 1$  and  $\text{NumCommLetts}(w_i, w_k) \neq j$  then
    counter  $\leftarrow$  counter + 1
  end if
end for
return counter
```

---

---

**Algorithm 4** AnswerProbs( $w, h, S$ )

---

```
for  $i = 0$  to  $L$  do
   $A_i \leftarrow 0$ 
end for
for  $i = 1$  to  $D$  do
  if  $S_i = 1$  then
     $k \leftarrow \text{NumCommLetts}(w, w_i)$ 
     $A_k \leftarrow A_k + h_i$ 
  end if
end for
Normalize  $A$  so its elements sum to 1.
return  $A$ 
```

---

---

**Algorithm 5** NumCommLetts( $w_i, w_j$ )

---

```
counter  $\leftarrow 0$ 
for  $m = 1$  to  $L$  do
   $c_1 \leftarrow m$ th character of  $w_i$ 
  for  $n = 1$  to  $L$  do
     $c_2 \leftarrow n$ th character of  $w_j$ 
    if  $c_1 = c_2$  then
      counter  $\leftarrow$  counter + 1
    end if
  end for
end for
return counter
```

---

## 6.2 Computing the hider's best response

In order to compute the hider's best response (in the context of fictitious play), we will find it useful to introduce two data structures. Let *IterNumGuesses* (ING) and *AvgNumGuesses* (ANG) be two arrays of size  $D$ . The  $i$ 'th component of ING will denote the number of guesses needed for the guesser's greedy best response to guess  $w_i$  at the current iteration of the algorithm. The  $i$ 'th component of ANG will be the average over all iterations (of fictitious play) of the number of guesses needed for the guesser's greedy best response to guess  $w_i$ . We update ANG by applying

$$ANG[i] = \left(1 - \frac{1}{t+1}\right) ANG[i] + \frac{1}{t+1} ING[i].$$

We update ING at each time step by applying

$$ING = \text{CompNumGuesses}(s_{h,t}),$$

where  $s_{h,t}$  is the hider's strategy at iteration  $t$  of fictitious play, and pseudocode for *CompNumGuesses* is given below in Algorithm 6. *CompNumGuesses* computes the number of guesses needed for the guesser's greedy best response to  $s_{h,t}$  to correctly guess each word. It accomplishes this by repeatedly querying *GuesserGBR* at various game states. The subroutine *UpdateState* updates the game state in light of the answer received from *GuesserGBR*. It is in this way that the hider's best response algorithm selectively queries the guesser's strategy, which is represented implicitly in oracular form.

Finally, once ING and ANG have been updated as described above, we are ready to compute the full best response for the hider. Pseudocode for the algorithm *HiderBR* is given below in Algorithm 8. *HiderBR* takes ANG as input,



---

**Algorithm 6** CompNumGuesses( $s_h$ )

---

```

for  $i = 1$  to  $D$  do
   $S \leftarrow$  vector of size  $D$  of all ones.
   $numguesses \leftarrow 0$ 
  while TRUE do
     $numguesses \leftarrow numguesses + 1$ 
     $nextguess \leftarrow GuesserGBR(s_h, S)$ 
     $answer \leftarrow NumCommLetts(w_i, nextguess)$ 

    if  $answer = L$  then
      BREAK
    end if
     $S \leftarrow UpdateState(S, nextguess, answer)$ 
  end while
   $x_i \leftarrow numguesses$ 
end for
return  $x$ 

```

---



---

**Algorithm 7** UpdateState( $S, nextguess, answer$ )

---

```

for  $i = 1$  to  $D$  do
  if  $S_i = 1$  and  $NumCommLetts(nextguess, w_i) \neq answer$  then
     $S_i \leftarrow 0$ 
  end if
end for
return  $S$ 

```

---



---

**Algorithm 8** HiderBR(ANG)

---

```

 $x^* \leftarrow \max_i ANG[i]$ .
 $T^* \leftarrow \{i : ANG[i] = x^*\}$ 
for  $i = 1$  to  $D$  do
  if  $ANG[i] = x^*$  then
     $y_i = \frac{1}{|T^*|}$ 
  else
     $y_i = 0$ 
  end if
end for
return  $y$ 

```

---

and determines which word(s) required the most guesses on average. If there is a unique word requiring the maximal number of guesses, then that word is selected with probability 1. If there are multiple words requiring the maximal number of guesses, then these are each selected with equal probability. Note that selecting any distribution over these words would constitute a best response; we just choose one such distribution. The asymptotic running time of CompNumGuesses is  $O(D^4L)$ , while that of HiderBR is  $O(D)$ .

Note that, unlike GuesserGBR of Section 6.1 which is an approximate best response using 1-ply minimax search, HiderBR is a full best response. We are able to compute a full best response for the hider because his strategy space is much smaller than that of the guesser; the hider has only  $D$  possible pure strategies — 2833 in the case of 5-letter Jotto.

### 6.3 Parallelizing the best response calculation

The hider’s best response calculation can be sped up dramatically by parallelizing over several cores. In particular, we parallelize the CompNumGuesses subroutine as follows. For the first processor, we iterate over  $i = 1$  to  $\lfloor \frac{D}{P} \rfloor$ , where  $P$  is the number of processors, and so on for the other processors. Thus we can perform  $P$  independent computations in parallel. The overall running time of the new algorithm is  $O\left(\frac{D^4L}{P}\right)$ .

## 7 Computing an approximate equilibrium in Jotto

We would like to apply smoothed fictitious play to Jotto, using HiderBR for the hider’s best response and GuesserGBR for the guesser’s best response. Unfortunately, this is tricky for a few reasons. First, it is not clear how to compute

---

**Algorithm 9** HiderBRPayoff(ANG)

---

```
maxnumguesses  $\leftarrow$  0
for  $i = 0$  to  $D$  do
  if ANG[i] > maxnumguesses then
    maxnumguesses  $\leftarrow$  ANG[i]
  end if
end for
return maxnumguesses
```

---

---

**Algorithm 10** HiderActualPayoff(ANG,  $s_h$ )

---

```
result  $\leftarrow$  0
for  $i = 0$  to  $D$  do
  result  $\leftarrow$  result + ANG[i]  $\cdot$   $s_h[i]$ 
end for
return result
```

---

---

**Algorithm 11** GuesserBRPayoff(ING,  $s_h$ )

---

```
result  $\leftarrow$  0
for  $i = 0$  to  $D$  do
  result  $\leftarrow$  result + ING[i]  $\cdot$   $s_h[i]$ 
end for
return -1  $\cdot$  result
```

---

---

**Algorithm 12** GuesserActualPayoff(ANG,  $s_h$ )

---

```
return -1  $\cdot$  HiderActualPayoff(ANG,  $s_h$ )
```

---

the epsilons and determine the quality of our strategies. And second, it will be difficult to run the algorithm without explicitly represent the guesser’s strategy; furthermore, we cannot output it at the end of the algorithm.

It turns that using the data structures developed in Section 6.2, we can actually compute the epsilons quite easily. This is accomplished using the procedures given in Algorithms 9– 12.

We are now ready to present our full algorithm for computing an approximate equilibrium in Jotto; pseudocode is given in Algorithm 13. Note that we initialize the hider’s strategy to choose each word uniformly at random. In terms of the guesser’s strategy, it turns out that all the information needed to obtain it is already encoded in the hider’s strategy and that we do not actually need to represent it in the course of algorithm.

To obtain the guesser’s final strategy, note that the strategies of the hider are output to a file at each iteration. It turns out that we can use this file to efficiently generate samples from the guesser’s strategy, even though we never explicitly output this strategy. We present pseudocode in Algorithm 14 for generating a sample of the guesser’s strategy at state  $S$  from the file output in Algorithm 13. This algorithm works by randomly selecting an integer  $t$  from 1 to  $t^*$ , then playing the guesser’s greedy best response to  $s_{h,t}$  — the hider’s strategy at iteration  $t$  of the algorithm. We can view this algorithm as representing the guesser’s strategy as a mixed oracular strategy; in particular, it is the uniform distribution over his greedy best responses in the first  $t^*$  iterations of Algorithm 13. This is noteworthy since it is a rare case of the mixed strategy representation having a computational advantage over the behavioral strategy representation.

## 8 Results

We ran our algorithm SolveJotto on four different Jotto instances, allowing words to be 2, 3, 4, or 5 letters long. To speed up the computation, we used the parallel version of the bottleneck subroutine CompNumGuesses (described in Sec-

---

**Algorithm 13** SolveJotto( $T$ )

---

$s_{h,0} \leftarrow (\frac{1}{D}, \dots, \frac{1}{D})$   
 Output  $s_{h,0}$  to StrategyFile  
 $ING \leftarrow \text{ComputeNumGuesses}(s_{h,0})$   
 $ANG \leftarrow ING$   
 Compute  $\epsilon$ 's per Algorithms 9-12,  $t^* \leftarrow 0$   
**for**  $t = 1$  to  $T$  **do**  
    $s_{h,t}^{BR} \leftarrow \text{HiderBR}(ANG)$   
    $s_{h,t} \leftarrow (1 - \frac{1}{t+1}) s_{h,t-1} + \frac{1}{t+1} s_{h,t}^{BR}$   
   Output  $s_{h,t}$  to StrategyFile  
    $ING \leftarrow \text{ComputeNumGuesses}(s_{h,t})$   
    $ANG \leftarrow (1 - \frac{1}{t+1}) ANG + \frac{1}{t+1} ING$   
   Compute  $\epsilon$ 's per Algorithms 9-12  
   **if**  $\epsilon < \epsilon^*$  **then**  
      $\epsilon^* \leftarrow \epsilon$ ,  $t^* \leftarrow t$   
   **end if**  
**end for**  
**return**  $(s_{h,t^*}, t^*, \text{StrategyFile})$

---



---

**Algorithm 14** ObtainGuesserStrategy( $\text{StrategyFile}, t^*, S$ )

---

$i \leftarrow \text{randint}(1, t^*)$   
 $s_h \leftarrow$  strategy vector on  $i$ 'th line of StrategyFile  
**return** GuesserGBR( $s_h, S$ )

---

tion 6.3) with 16 processors. As our dictionary, we use the Official Tournament and Club Word List [8], the official word list for tournament Scrabble in several countries. As discussed in Section 4, we omit words with duplicate letters and words for which there exists an anagram that is also a word. The dictionary sizes are given in Table 1. We note that our algorithms extend naturally to any number of words and dictionary sizes (and to other variants of Jotto as well).

Number of letters	2	3	4	5
Dictionary size	51	421	1373	2833
Our hider payoff vs. benchmark	7.652	7.912	7.507	7.221
Our guesser payoff vs. benchmark	-6.619	-7.635	-7.415	-7.216
Our overall payoff vs. benchmark	0.517	0.139	0.046	0.003
Benchmark self-play hider payoff	6.627	7.601	7.365	7.079
Our algorithm self-play hider payoff	7.438	7.658	7.390	7.162
Benchmark epsilon	5.373	3.399	1.635	1.921
Our final epsilon	0.038	0.334	0.336	0.335
Number of iterations	22212	10694	3568	3906
Avg time per iteration (minutes)	$3.635 \times 10^{-4}$	0.028	1.160	12.576

**Table 1.** Summary of our experimental results.

One metric for evaluating our algorithm is to play the strategies it computes against a benchmark algorithm. The benchmark algorithm we chose selects his word uniformly at random as the hider, and plays the greedy best response to the uniform strategy as the guesser. This is the same strategy that we use to initialize our algorithm.

For each number of letters, we computed the payoff of our algorithm SolveJotto against the benchmark. Recall that the payoff of the hider is the expected number of guesses needed to correctly guess the word (and the guesser's payoff is -1 times the hider's payoff). The overall payoff is the average of the hider and guesser payoff.

Several observations from Table 1 are noteworthy. First, our algorithm beats the benchmark for all dictionary sizes. In the two-letter game, our expected payoff against the benchmark is 0.517; our strategy requires over a full guess less than the benchmark in expectation. Our profit against the benchmark decreases as more letters are used.

In addition to head-to-head performance against the benchmark, we also compared the algorithms in terms of worst-case performance. Recall that  $\epsilon$  denotes the maximum payoff improvement one player could gain by deviating to a best response (full best response for the hider and greedy best response for the guesser). Note that in all cases, our  $\epsilon$  is significantly lower than that of the benchmark. For example, in the two-player game the benchmark obtains an  $\epsilon$  of 5.373, while our algorithm obtains one of 0.038.

Interestingly, we also observe that the self-play payoff of our algorithm, which is an estimate of the value of the game, does not increase monotonically with the number of letters. That is, increasing the number of letters in the game does not necessarily make it more difficult for the guesser to guess the hider’s word.

## 9 Conclusion

We presented a new approach for computing approximate-equilibrium strategies in Jotto. Our algorithm produces strategies that significantly outperform a benchmark algorithm with respect to both head-to-head performance and worst-case exploitability. The algorithm extends fictitious play to a novel strategy representation called oracular form. We expect our algorithm and the oracular form representation to apply naturally to many other interesting games as well.

## References

1. Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., Szafron, D.: Approximating game-theoretic optimal strategies for full-scale poker. In: IJCAI (2003)
2. Brandt, F., Fischer, F., Harrenstein, P.: On the rate of convergence of fictitious play. In: SAGT (2010)
3. Brown, G.W.: Iterative solutions of games by fictitious play. In: Koopmans, T.C. (ed.) *Activity Analysis of Production and Allocation* (1951)
4. Ganzfried, S., Sandholm, T.: Computing an approximate jam/fold equilibrium for 3-player no-limit Texas Hold’em tournaments. In: AAMAS (2008)
5. Harsanyi, J.: Games with incomplete information played by Bayesian players. *Management Science* (1967)
6. Jotto entry in wikipedia, <http://en.wikipedia.org/wiki/Jotto>
7. Rabinovich, Z., Gerding, E., Polukarov, M., Jennings, N.R.: Generalised fictitious play for a continuum of anonymous players. In: IJCAI (2009)
8. Official tournament and club word list, <http://www.isc.ro/en/commands/lists.html>
9. Zhu, W., Wurman, P.: Structural leverage and fictitious play in sequential auctions. In: AAAI (2002)
10. Zinkevich, M., Bowling, M., Johanson, M., Piccione, C.: Regret minimization in games with incomplete information. In: NIPS (2007)